

DESES: Middleware for Dynamic End-to-end Session-Enhancing Services for Java-enabled Mobile Phones

Dimitris N. Kalofonos
Pervasive Computing Group
Nokia Research Center Cambridge
Cambridge, MA 02142, U.S.A.
dimitris.kalofonos@nokia.com

Parijat Shah
College of Engineering
Northeastern University
Boston, MA 02115, U.S.A.
parijat@coe.neu.edu

Elias S. Manolakos*
ECE Department
Northeastern University
Boston, MA 02115, U.S.A.
elias@ece.neu.edu

Abstract—Networking middleware approaches that offer end-to-end (e2e) TCP/IP session enhancements are gaining importance in the Internet research community. Examples of such session enhancements include protection from disconnections, mobility support, congestion management, encryption and compression. These session-enhancing frameworks require only middleware support at the end-nodes, thus making them ideal for dynamic environments where there is little or no infrastructure support. Mobile devices would benefit the most from such frameworks, especially if they were dynamic, extensible and platform-independent. In this paper we describe such a framework optimized for mobile devices called DESES: Dynamic End-to-end Session-Enhancing Services for Java-enabled mobile devices. DESES enhances networked J2ME applications by allowing the dynamic addition and removal of e2e session-enhancing services during the lifetime of an active TCP/IP session. We describe the DESES architecture and how DESES services can be deployed transparently, if desired, on existing J2ME applications using a technique called byte-code engineering. Finally, we describe our experience implementing DESES on actual J2ME-enabled smartphones.

I. INTRODUCTION

Mobile devices operating in pervasive computing environments are faced with difficult connectivity conditions. Common problems arise from signal fluctuations, the limited range of proximity wireless technologies, mobility and roaming between access points and/or different access technologies, connectivity through unreliable ad-hoc networks, limited bandwidth and high round-trip delays. The performance of TCP/IP sessions can be severely affected by the prevalent conditions in such hostile environments, impacting in-turn the distributed applications running on top of the TCP/IP protocol suite. The end-users often experience slow and intermittent data transfers, network disconnections and annoying application errors which result in their frustration. In most cases, networked applications do not get support either from the operating system or from the native networking protocols to handle these conditions; it is left to application developers to deal with these problems, who often do not anticipate them. A better approach is to develop middleware that provides system-level support to mobile applications in the form of session-enhancing services. Examples of such session-enhancing services include mobility

support to handle network disconnections and changes of IP addresses, compression over low-bandwidth links and encryption services for securing network flows in insecure networks.

A large number of academic research efforts and industry initiatives attempt to address many of these problems by providing enhancements at different levels: link-level, network-level, transport-level, and session-level. Even though many of the above approaches provide some of the improvements/enhancements needed, they often do so by requiring infrastructure support, e.g. dedicated servers, proxies, access point coordination. This requirement makes most of these approaches inapplicable in cases where such infrastructure is not present, e.g. in ad-hoc scenarios and unmanaged home networks. More recently, a number of approaches have been proposed which provide end-to-end mobility support and support against disconnections at the socket-level, e.g. MobileSocket [1], ROCKS/RACKS [2], Mobile TCP Sockets [3], TESLA [4], Migrate [5]. These approaches provide the applications with a virtual socket interface, which allows actual network sockets to be destroyed and replaced as necessary to deal with IP address changes and disconnections resulting in TCP expirations. These approaches do not require any support from the infrastructure and any intermediate nodes, which makes them particularly suitable for pervasive computing environments. However, none of these approaches was designed and optimized specifically for limited-resource devices such as mobile phones.

This paper presents DESES, a middleware providing Dynamic End-to-end Session Enhancing Services for Java-enabled mobile phones. Mobile phones use a wide variety of operating systems, often proprietary, therefore it is important to choose a mobile platform-independent approach, e.g. based on the ubiquitous Java Mobile Edition (J2ME). The DESES architecture is based on the concept of virtual socket middleware, in particular as this was proposed in TESLA [4] and Migrate [5]. We modified the Linux-based TESLA architecture to target Java phone platforms and extended it to make it more dynamic in managing sessions. To the best of our knowledge, DESES is the first such middleware to enable development and deployment of e2e session-enhancing services for J2ME applications. Other contributions of DESES and this paper include: a technique to offer transparent session support for

*Current address: Dept. of Informatics and Telecommunications, National and Kapodistrian University of Athens, Greece.

legacy J2ME applications without requiring any modifications; a session negotiation and a session renegotiation mechanism that lets users/agents on the two end-nodes to select the kind of e2e services to be used and later to change them on-the-fly if necessary; finally, the experience of prototyping on actual mobile phones that support J2ME MIDP v2.0.

The rest of this paper is organized as follows: Section II presents a review of related work; Section III presents a motivating usage scenario and the requirements we considered in the design of DESES; Section IV describes in detail the DESES architecture and the process of session negotiation and renegotiation; Section V presents details about the DESES J2ME library implementation and byte-code engineering, the technique applied to transparently interpose the DESES J2ME libraries; Section VI presents our experience with DESES on actual smartphones; finally, Section VII gives our conclusions.

II. RELATED WORK

Solutions to the problems of mobility (link-level and IP-level) and TCP performance optimization include: GSM mobility [6], mobile IP [7], TCP Optimizations for Wireless [8], TCP-R [9], MSOCKS [10], Session Layer Mobility (SLM) [11] and SIP-based mobility [12]. These approaches require infrastructure support (proxies, dedicated servers, access point collaboration) and/or OS kernel modifications and recompilation of networking protocols. These requirements make these solutions fairly static and often inapplicable in dynamic infrastructure-less environments.

A different approach is to isolate applications from network problems by providing a virtual socket interface. Such frameworks provide session enhancements such as IP mobility support and protection from TCP expiration without requiring support from intermediate nodes. An example of such frameworks is Reliable Sockets (ROCKS) [2] that provides support against network disconnections to native applications on Unix/Linux platforms by interposing its libraries between the applications and the native system networking support. ROCKS is platform-dependent and makes some assumptions about application behavior that may limit its applicability in some cases.

Another example is TESLA [4], that provides a comprehensive framework for session-enhancing services for native applications on Linux platforms. Similarly to ROCKS, it does so by linking itself to the native applications at run-time and interposing itself between an application and the native system networking support. TESLA is extensible, allowing a multitude of e2e services such as mobility support (see Migrate [5]), e2e compression and encryption. Although extensible, TESLA requires prior knowledge of the e2e services to be used before starting an application; once it is started no services can be added or removed, which makes it rather static. Furthermore, TESLA is Unix/Linux platform-dependent. DESES aims at enhancing TESLA by making it (a) more dynamic with negotiation of the services to be used at session establishment and renegotiation at any time during an active session; and (b)

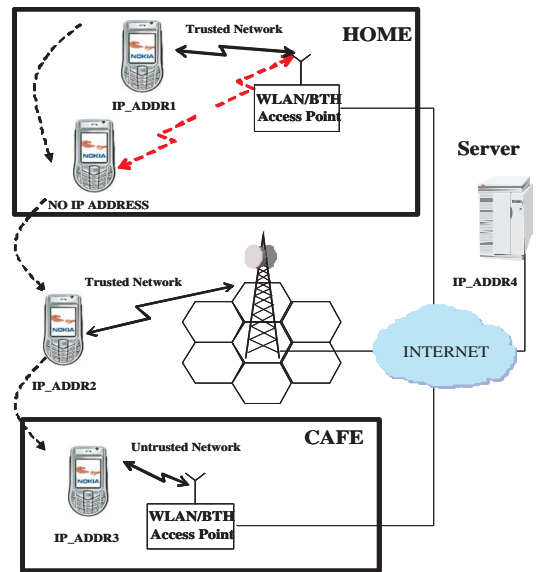


Fig. 1. A usage scenario illustrating functionality offered by DESES.

more platform-independent, suitable for the pervasive Java-enabled mobile devices.

A proposal based on the concept of virtual sockets for Java J2SE applications is MobileSocket [1]. MobileSocket provides Java applications mobility support against changes of IP addresses and network disconnections through an API similar to the Socket API that Java developers can use when developing networked applications. However, its architecture is not flexible nor dynamic and does not allow the development and deployment of any other session-enhancing services.

III. USE CASES AND DESIGN REQUIREMENTS

A. Motivating Usage Scenario

To provide the motivation behind this work, consider the following usage scenario illustrated in Figure 1. While at home, a user starts a background session to synchronize her mobile phone's music collection with her service provider's music server. Both her phone and the remote server feature middleware that provides dynamic e2e session-enhancing services. The mobile phone is connected to the Internet through her home's secure WLAN access point. A middleware agent on her phone starts the session after negotiating with the remote server's agent the use of e2e mobility support, but neither e2e compression nor e2e encryption, in order to reduce local computation and save battery. At some point she walks out of the building and gets disconnected. The synchronization pauses, but there are no errors because the e2e mobility support service suspends the session transparently to the application. When outside the building her phone switches to the secure but low-bandwidth GPRS network. The phone's agent detects that connectivity is restored and renegotiates with the other end the e2e services to be used based on the new conditions. They agree to add e2e compression to deal with the low bandwidth and they resume the session. After a while the user sits at

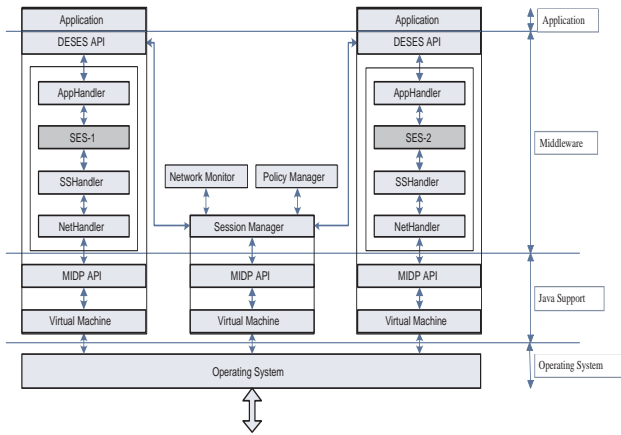


Fig. 2. Overview of the DESES end-node architecture.

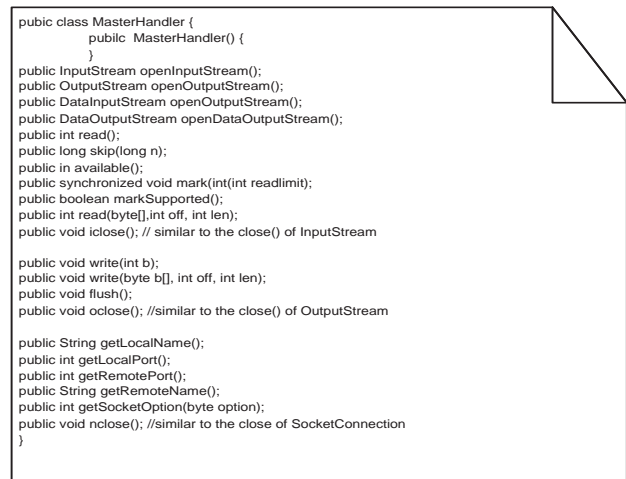


Fig. 3. Definition of the *MasterHandler* class.

a cafe, where there is a WLAN access point available for public use. The agent in her mobile phone detects this and suspends the session to reconnect through the high-bandwidth WLAN access point. The two agents renegotiate to remove e2e compression and add e2e encryption and the session resumes until synchronization completes.

B. Design Requirements

Motivated by the above scenario and our goal for DESES to be as widely applicable to the mobile world as possible, we pose the following design requirements for DESES: it should provide end-to-end session enhancing services without requiring infrastructure support; it should be based on the pervasive Java Mobile Edition (J2ME) for mobile platform independence; it should provide enhancements to legacy applications as transparently as possible; it should provide an easy to use API for developers to develop new distributed applications; it should be extensible to accommodate new session-enhancing services; it should be able to negotiate which, if any, e2e session-enhancing services will be used at session establishment; finally, it should provide a mechanism to dynamically add and remove session-enhancing services at any time during an active session.

IV. SYSTEM ARCHITECTURE AND DESIGN

A. Overview of the DESES Architecture

As mentioned earlier, the DESES architecture is based on the concept of virtual sockets, in particular as this was proposed in TESLA [4] and Migrate [5]. DESES follows a flow-based abstraction to session-enhancing services. Figure 2 depicts an overview of the DESES architecture. As seen in Figure 2, a stack of DESES handlers are interposed between the application and the MIDP API for network support. Data flows between the application and the actual network sockets through these handlers, each of which performs certain functionality and passes the data to the following one. Eventually, data is written to the network through the native socket interface and, on the opposite direction, received data is consumed by the application. DESES provides a flexible architecture that

does not restrict the number and type of handlers to be used, as long as both ends support and agree to use the same set of handlers. However, there are at least three handlers that are always present: the application handler (*AppHandler*), the network handler (*NetHandler*) and the session support handler (*SSHHandler*). Section IV-B describes these handlers in detail.

In contrast to TESLA which proposes a separate process for the stack of data handlers, the DESES handlers are part of the application/virtual-machine process itself. We chose not to follow the TESLA two-process approach (application + handlers) because of the difficulties posed by the J2ME API, which does not provide means to know when the TCP send buffer is full. This would prevent the application thread from being blocked when it tried to write data that the handlers could not handle. Besides the application/handlers process, the DESES architecture includes two more modules running as separate processes and offering their services to all running DESES-enabled applications: the *Session Manager* and the *Network Monitor*. The *Session Manager* negotiates the kind of session enhancing services to be used and facilitates any changes of these services while a session is active. The *Network Monitor* informs the *Session Manager* about any changes to the node's connectivity status. Interprocess communications among the different DESES processes is achieved using regular sockets and the loopback interface. Finally, DESES assumes the presence of some kind of *Policy Manager*, which acts as an agent that decides about what kind of session-enhancing services are to be used based on the prevailing conditions. However, the definition of the *Policy Manager* is left out of the scope of this paper.

B. The DESES Handlers

Each session-enhancing service is implemented as an instantiation of a **MasterHandler** class. Each *MasterHandler* object takes as input a single network flow, and produces a single network flow as output. The *MasterHandler* class defines the methods defined in the *javax.microedition.io.SocketConnection* interface,

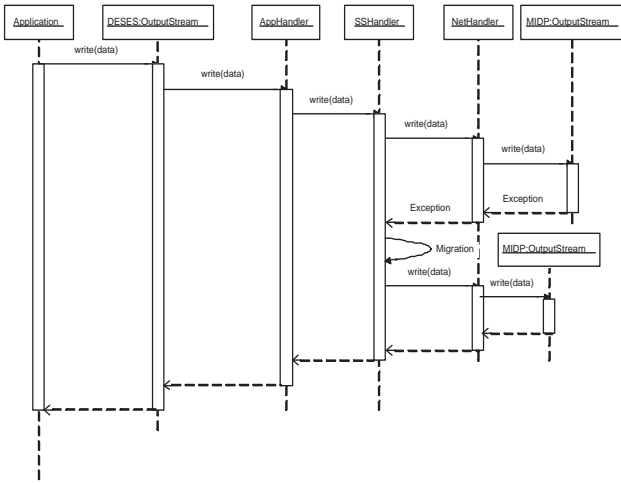


Fig. 4. UML sequence diagram showing the role of the *SSHHandler*.

javax.microedition.io.ServerSocketConnection interface, *java.io.OutputStream* class and *java.io.InputStream* class. The method semantics of these methods are the same as those defined in the corresponding J2ME classes and interfaces. The definition of the *MasterHandler* class can be seen in Figure 3.

At session establishment, the *Session Manager* negotiates the list of session-enhancing services to be used and passes them to the DESES library which is part of the application process. DESES then instantiates the handlers implementing these session-enhancing services and plugs them in as nodes in an ordered list. The handlers operate on each network flow in the order that was negotiated. For any handler, a handler immediately below is called “downstream” handler and a handler immediately above it is called “upstream” handler. By default all the methods in the *MasterHandler* class are implemented to delegate method calls to its downstream handler.

The *AppHandler* acts as an interface between the DESES API and the first session-enhancing service. The DESES API delegates all the method calls that the application invokes to the *AppHandler*. The *AppHandler* itself does not define any of these methods and inherits them from the *MasterHandler*. The downstream handler of the *AppHandler* is always the first session-enhancing service.

The *NetHandler* is always the last handler that operates on the network flow. While the application reads and writes data using the virtual sockets that DESES exposes, it is the *NetHandler* that uses actual network sockets to send and receive data. The *NetHandler* overrides all the methods defined in the *MasterHandler* class and performs input/output operation on the *InputStream* and the *OutputStream* corresponding to the connected socket. One can think of the *NetHandler* as an interface between DESES and the networking support of Java.

The *SSHHandler* is always the upstream handler of the *NetHandler* and delegates the method calls to it. In the regular J2ME platform, an application that attempts to read

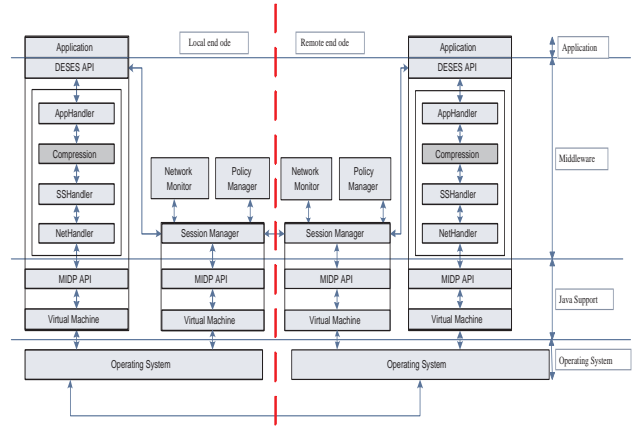


Fig. 5. End-to-end communication of two DESES enabled applications.

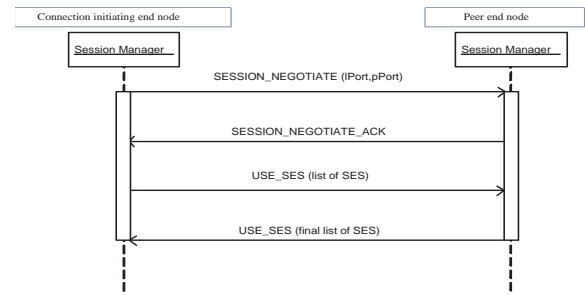


Fig. 6. UML sequence diagram illustrating the process of session negotiation.

or write data to an expired connection receives an exception. In DESES, the *SSHHandler* catches all the exceptions caused by network errors (e.g. TCP expirations) and isolates the application from them. Because data may be lost during connection failures, if a reliable connection is needed (e.g. TCP), it is necessary to buffer all the in-flight data for possible re-transmission [1], [2], [5]. The *SSHHandler* blocks in the *write()* or *read()* method that results in exception, thus isolating the application from the error. When a new TCP connection is formed to replace the failed TCP connection, the lost in-flight data is retransmitted. Then the application thread, which had been blocked in the method that resulted in the exception, resumes without receiving an exception. Figure 4 illustrates how the *SSHHandler* protects the application from network failures.

C. Dynamic Session Management

The *Session Managers* at the two end-nodes collaborate to provide two of the most important functionalities of DESES: session negotiation and renegotiation. Figure 5 depicts this e2e interaction between the DESES middleware at the local and the remote node.

Session Negotiation: two DESES-enabled end-nodes may not support or may not be willing to use the same set of session-enhancing services. The *Session Managers* perform the task of negotiating which services will be used. Figure 6 depicts the process followed by the two *Session Managers*.

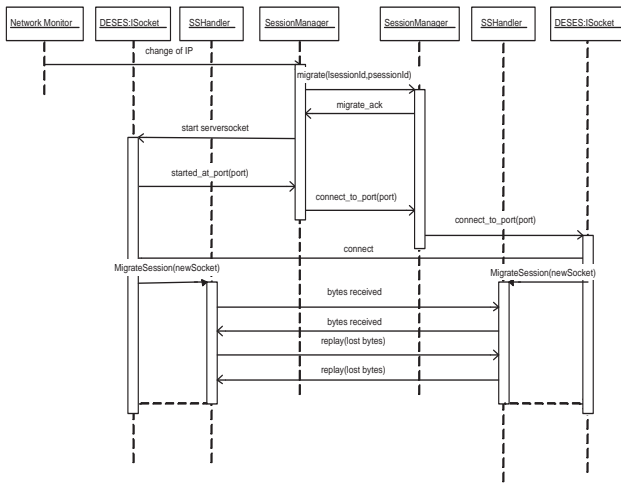


Fig. 7. UML sequence diagram illustrating the process of session resumption.

The *Session Manager* on the connection-initiating end-node first sends to the remote *Session Manager* the local port number (IPort) and remote port number (pPort) to identify the connection for which to perform session negotiation. After receiving an acknowledgment, the first *Session Manager* sends a list of services it would like to use in the new session. The second *Session Manager* sends back its own list of services and this process may go through a limited number of iterations until the two *Session Managers* agree on the same list. Of course, each *Session Manager* may decide to abort the connection if the other end does not accept the session-enhancing services required by its *Policy Manager*. Once the session negotiation completes, the *Session Manager* on each end sends the list of agreed upon services to the DESES library, which instantiates an ordered list of corresponding handlers.

Session Suspension and Resumption: a session is suspended when a connection is broken (e.g. TCP expires, IP connectivity is lost). After IP connectivity is restored, the *Session Manager* receives an event from the *Network Monitor* and initiates a new TCP/IP connection. Figure 7 illustrates the steps involved to resume the session and restore application communication. The *Session Manager*, after receiving notification by the *Network Monitor* about the new IP address upon reconnection, sends to the remote *Session Manager* the local (lsessionId) and remote (psessionId) *sessionId*. The *sessionId* uniquely identifies a session on each end-node. If the remote *Session Manager* finds the session, it sends an acknowledgement. The local *Session Manager* upon receiving the acknowledgement asks the DESES *ISocket* library to start a server socket. The *ISocket* starts the server socket and sends the port number to which it is listening to the local *Session Manager*, which in turn sends the port number to the remote *Session Manager*. The remote *Session Manager* sends the port number to its *ISocket* library, which opens a new TCP/IP connection to this port. This newly formed connection is used to replace the expired connection and to resume application

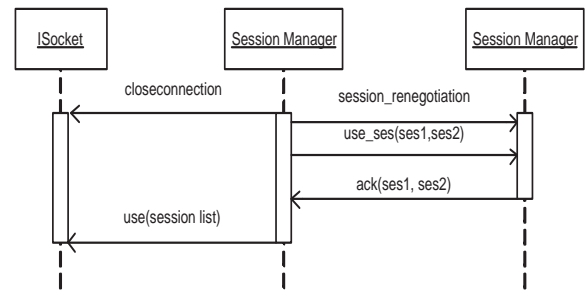


Fig. 8. UML sequence diagram illustrating the process of session renegotiation.

communication. Before application communication resumes, each end-node exchanges the number of bytes received and replays the lost data.

Session Renegotiation: in contrast to other virtual socket frameworks [1], [2], [5], [4] which are mostly static, DESES proposes a dynamic process to renegotiate the list of used session-enhancing services at any time during an active session. The session renegotiation process is triggered by the *Session Manager* every time just before a session resumes from a disconnection. As described earlier, this disconnection is usually caused by mobility or intermittent connectivity. However, DESES also allows the *Session Manager* to break an active connection on purpose, in order to renegotiate the e2e services used upon its resumption. Changing e2e services would lead to corrupted data unless both sides coordinate carefully. Figure 8 depicts the process of the session renegotiation. The *Session Manager* that initiates it sends a message to its *ISocket* library to close the existing TCP/IP connection and then sends a new proposed list of session-enhancing services to be used to the remote *Session Manager*. The two *Session Managers* again negotiate and agree on the new list of e2e services. The *SSHHandler* isolates the closing of the connection from the application as described earlier. The process of resuming the session after session renegotiation is completed is the same as described earlier and shown in Figure 7.

V. IMPLEMENTATION OF THE DESES J2ME LIBRARY

A. The DESES Library API

Mobile phones use the Mobile Information Device Profile (MIDP) of J2ME built on the Connected Limited Device Configuration (CLDC). The Generic Connection Framework [13] provides the interface hierarchy for connectivity. At the top of the interface hierarchy is the *Connection* interface, the most basic type of connection. All other connection types extend the *Connection* interface. MIDP v2.0 introduces the *ServerSocketConnection*, *HttpConnection*, *UDPDataConnection* and *SocketConnection* interfaces. The generic connection framework provides a connection factory by providing the *Connector* class. All connection types are opened using the static *open()* method of the *Connector* class. The Uniform Resource Locators indicate the kind of connection to open.

Figure 9 shows the interface and class hierarchy of the DESES Library API. The DESES API provides the wrapper

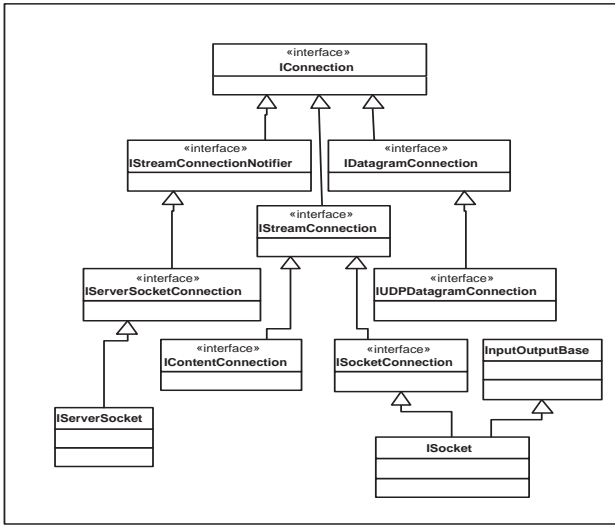


Fig. 9. Interface and class hierarchy of the DESES API.

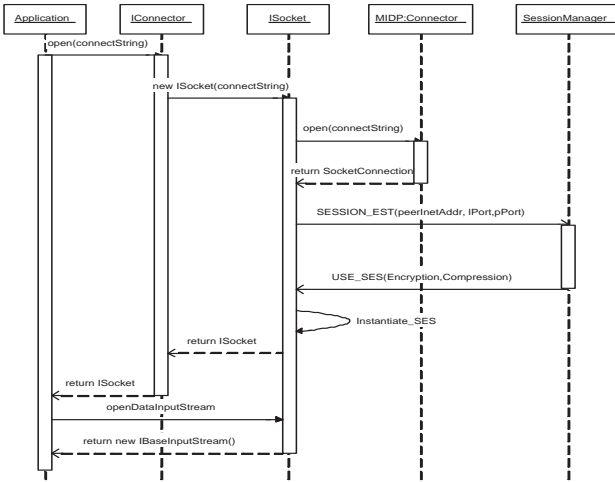


Fig. 10. UML sequence diagram depicting the use of the *ISocket* constructor.

classes *ISocket* and *IServerSocket* corresponding to MIDP v2.0 classes for client and server sockets respectively. The semantics of the methods defined in the wrapper classes is similar to those of the corresponding J2ME classes. DESES also introduces the *IConnector* class, similar to the *Connector* class. When the application invokes the static *open()* method of the *IConnector* class, the instance of the *ISocket* is returned. Figures 10 and 11 show what happens when the *ISocket* and *IServerSocket* constructors are invoked.

To intercept the application's communications, DESES defines the classes *IBaseInputStream* and *IBaseOutputStream* to extend the *InputStream* and *OutputStream* abstract classes respectively, which are defined in Java as the superclasses of all classes representing input and output streams of bytes. When a DESES-enabled application requests an *InputStream* or an *OutputStream*, DESES returns the objects of the classes *IBaseInputStream* and *IBaseOutputStream*. Therefore, when the application performs any input-output operations, the

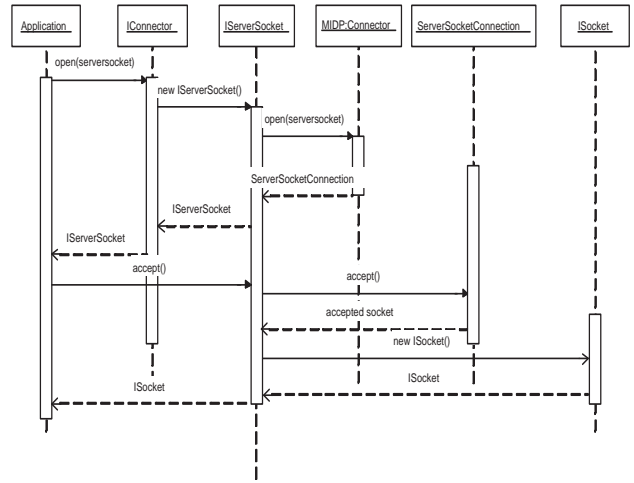


Fig. 11. UML sequence diagram depicting the use of the *IServerSocket* constructor.

read() and *write()* methods of *IBaseInputStream* and *IBaseOutputStream* get invoked. These methods are implemented to delegate *read()* and *write()* requests to the *AppHandler* of the DESES middleware described in Section IV-B.

B. Byte-Code Engineering

One of the requirements for DESES is to provide support to legacy J2ME applications as transparently as possible. Middleware proposals such as [2] and [4] modify the LD_PRELOAD environment variable in Linux to interpose their libraries transparently. However, there is no such provision in Java to load classes other than those imported by applications. Instead, we propose to apply a technique called *byte-code engineering* [14], which modifies the Java class file before it is interpreted by the virtual machine. Using byte-code engineering one can modify the class file to replace e.g. the *java.microedition.io.Connector* class with an *IConnector* class. The virtual machine will then load the *IConnector* class and will invoke its *open()* method.

To show how byte-code engineering is performed, note that each Java class file contains a definition of a single class or an interface in the form of a stream of 8-bit bytes without any padding or spacing. Every class file consists of a single ClassFile structure whose fields are described in detail in the Java Virtual Machine (JVM) specification [15]. The Virtual Machine forms an instruction by referring to the class and interface names in the constant_pool. Byte-code engineering parses the class file and alters this instruction by modifying the string constants in the constant_pool.

As mentioned above, the DESES library API introduces the *IConnector* class and the *ISocketConnection* and *IServerSocketConnection* interfaces to replace the corresponding entities of the Generic Connection framework. Without byte-code engineering, adding DESES support to already developed applications would require modifications to the source code and re-compilation. As an alternative, we propose the technique of byte-code engineering described above to interpose the DESES

libraries transparently. This involves a pre-processing step where the application's byte-code class file is parsed and a new one is created with the DESES API class and interface names in the constant_pool. This pre-processing step can happen off-line during the process of installing the application in the mobile device or on-the-fly during the process of launching the application. It is possible that such modifications may require special security privileges; however, such privileges are usually available to mobile device manufacturers.

VI. A PROOF-OF-CONCEPT EXPERIMENT

As proof-of-concept, we conducted the experiment shown in Figure 12, where we used our DESES middleware implementation to provide support to a simple J2ME MIDP v2.0 TCP "file download" client application against network disconnections and IP address changes. In this experiment we used a Nokia 6600 Symbian-based smartphone and a Linux-based laptop, which were connected with Bluetooth using the PAN Profile to provide IP support. The size of the middleware to install on the phone was approximately 40 KB. We first developed the client application using the regular MIDP Generic Connection framework socket support. We then parsed the compiled class file with our byte-code engineering parser and we installed the DESES-enabled modified class file on the Nokia 6600. The server application was developed using the DESES API and was ran on the Linux laptop using the J2ME emulator. As shown in Figure 12, DESES provided support against the disconnection and IP address change transparently to the user, who only observed a pause in data transfer.

We would like to note that implementation on actual mobile phones presents additional challenges not encountered when using an emulator. The virtual machine used by the emulator is the standard JVM and not the KVM used in the actual mobile phones. It is possible, therefore, that a program that runs successfully on the emulator may not run on the actual mobile phone. Examples of experience gained through our prototype implementation on J2ME MIDP phones: a new thread must be created for every new connection opened or accepted; all exceptions must be carefully handled to avoid errors; finally, when writing data to the *OutputStream*, it is necessary to flush the data, otherwise it will not be written to the network.

VII. CONCLUSIONS

In this paper we described the design and implementation of DESES, a networking middleware offering e2e session-enhancing services to J2ME applications. At session establishment, DESES provides a mechanism to negotiate the session-enhancing services to be used. Furthermore, DESES introduces session renegotiation to dynamically change the session-enhancing services at any time during an active session. DESES provides a socket API similar to the J2ME MIDP v2.0 socket API, which can either be used directly to develop new DESES-enabled networked applications, or can be transparently interposed to existing applications using the byte-code engineering technique. Finally, we developed

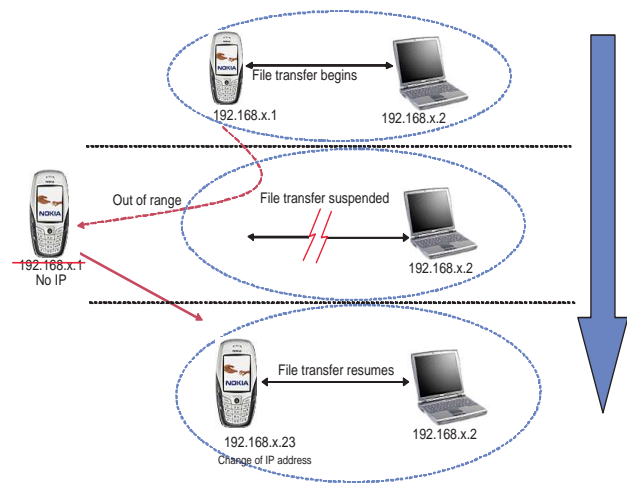


Fig. 12. Our proof-of-concept experiment.

a proof-of-concept implementation of DESES using actual J2ME-enabled mobile phones.

REFERENCES

- [1] Okoshi T., Mochizuki M., Tobe Y., and H.Tokuda. "MobileSocket: towards continuous operation for Java applications". In *8th IEEE International Conference on Computer Communications and Networks*, pages 50–57, October 1999.
- [2] Zandy V. and Miller B. "Reliable network connections". In *8th ACM International Conference on Mobile Computing and Networking (Mobicom'02)*, pages 95–106, September 2002.
- [3] Qu X., Xu Yu J., and Brent R. "A mobile TCP socket". In *International Conference on Software Engineering (SE97)*, November 1997.
- [4] Salz J., Snoeren A., and Balakrishnan H. "TESLA: a Transparent, Extensible Session-Layer Architecture for end-to-end network services". In *4th USENIX Symposium on Internet Technologies and Systems (USITS'03)*, pages 211–224, March 2003.
- [5] Snoeren A. "A Session-Based Architecture for Internet Mobility". Ph.D. thesis. MIT, Feb. 2003.
- [6] Schmid E. and Kahler M. "Overview of the GSM system and protocol architecture". *IEEE Communications Magazine*, 31(4):92–100, April 1993.
- [7] Perkins C. "RFC 3320 - IP mobility support for IPv4". Internet Engineering Task Force, January 2002.
- [8] Inamura H. et al. "RFC 3481 - TCP over Second (2.5G) and Third (3G) generation wireless networks". Internet Engineering Task Force, February 2003.
- [9] Funato D., Yasuda K., and Tokuda H. "TCP-R: TCP mobility support for continuous operation". In *IEEE International Conference on Network Protocols*, pages 229–236, October 1997.
- [10] Maltz D. and Bhagwat P. "MSOCKS: an architecture for transport layer mobility". In *IEEE Infocom'98*, pages 1037–1045, March 1998.
- [11] Landfeldt B., Larsson T., Ismailov Y., and Seneviratne A. "SLM, a framework for session layer mobility management". In *8th IEEE International Conference on Computer Communications and Networks (ICCCN99)*, October 1999.
- [12] Wedlund E. and Schulzrinne H. "Mobility support using SIP". In *2nd ACM Work. on Wireless Mobile Multimedia (WoWMoM'99)*, pages 76–82, August 1999.
- [13] "The generic connection framework", August 2003. <http://developers.sun.com/techtopics/mobility/midp/articles/genericframework>.
- [14] Dahm M. "Byte code engineering". In *Java-Informationen-Tage*, pages 267–277, September 1999.
- [15] Lindholm T. and Yellin F. "The Java Virtual Machine Specification" (2nd Edition). Addison-Wesley Professional, April 1999.